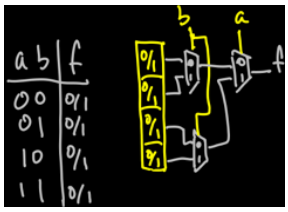
http://stpe.github.io/jniosemu/
setup time: time period before the clock posedge for which D has to be stable
hold time: time period after the clock posedge for which D has to be stable
clock to Q delay: delay time
minimum clock period Tmin= setup time+ hold time + clock to Q delay

Blocking: Q=D; effect of a statement is seen by following statements(inside an always block)
Non-Blocking: Q<=D; not seen. all statements are evaluated concurrently.

Field-Programmable Gate Array Architecture &Computer-Aided Design tools

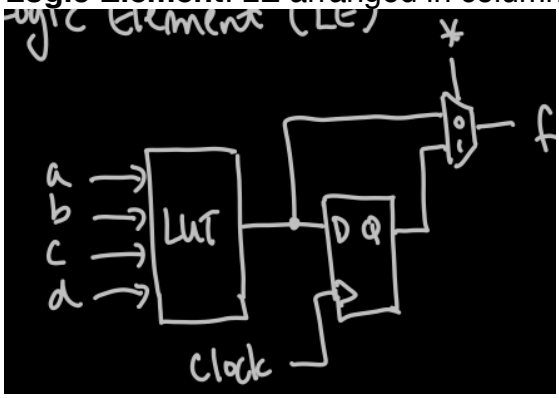
Look-Up Table: 2-inputs lookup table



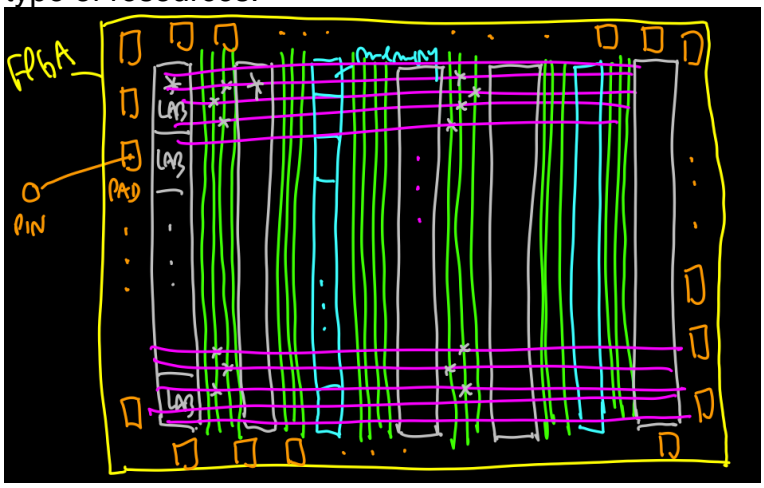
Circuit is broken up(decomposed) into functions of ≤4 inputs, and each functions truth table is stored in a LUT.

①find truth table(input a,b output f_1, f_2, f_3, f_4)
②transfer output column as LUT inputs; input a,b as select signal.

Logic Element: LE arranged in columns to be LAB

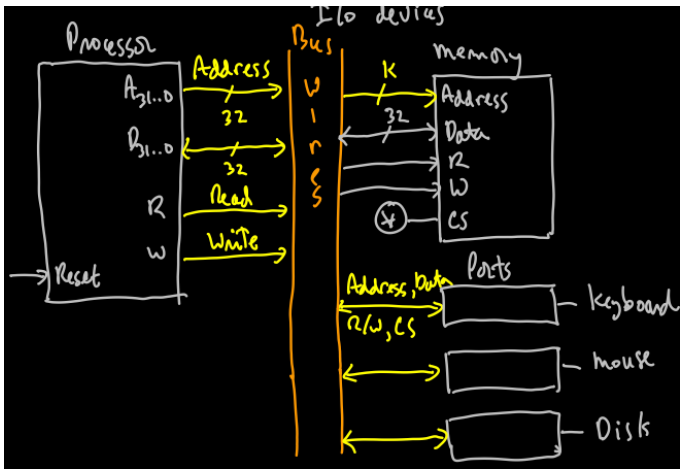


hierarchy(阶层)-- **logic array block**: LAB arranged in columns in the chip along with other type of resources.



CAD tools: Verilog>Synthesis>(LEs,FFs)>Placement(choose locations for LEs)>Routing(choose wires)>Programming(download to FPGA)

Computer Organization: processor, memory, program, I/O device

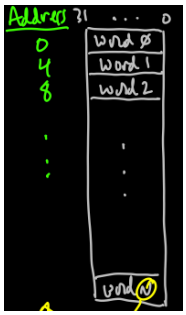


Chip Select: ensures that each address from the processor select exactly one device. (accomplished by address decoding)

chip select				to device			device
A_{31}	A_{30}	A_{29}	A_{28}	A_{27}	...	A_0	
0	0	0	0	x	...	x	memory
0	0	0	1	x		x	keyboard
0	0	1	0	x		x	mouse
..	..	..	..	..	...	..	...

Chip Select Logic: AND gate+NOT gate (4 input $A_{31} \sim A_{28}$)
Wires in Address Bus:
Wires in DataOut Bus: 一个wire 一个bit

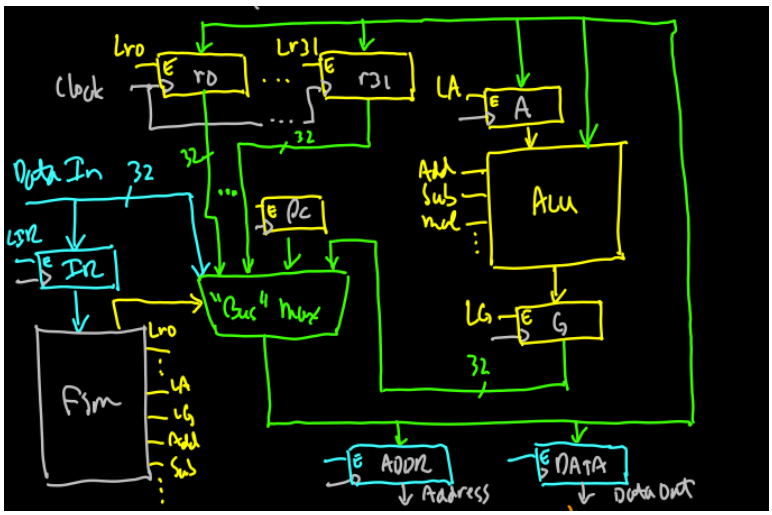
Memory Architecture



byte addressable, each word(4bytes,32bits) has 4 addresses.
memory k bits (2^k addresses), word from 0 to $2^{(k-2)}$-1
(previos k-2 bits select word,last two bits select bytes in a word)

Processor Architecture

Processor is a logic circuit that includes
①a set of registers(typically 32 bits), such as r0,...r31.
②an arithmetic and logic unit (ALU) that has adder/subtractor,multiplier, and can perform bit-wise logic operations(AND,OR,XOR),
③shifter... etc.
④an interface to memory(address/data)
⑤a finite state machine to control the processor.



program counter(PC): point to/hold address of next instruction to be executed. (++4)
Instruction Register(IR): hold instruction being executed currently.

add r0,r1,r2	# r2<-[r3]+[r4] [ ]: content of
Clock cycle 0: instruction is read from memory on DataIn; FSM set LIR Clock cycle 1: FSM sets select lines on Bus mux to choose r1, and set LA=1 Clock cycle 2: FSM sets select lines on Bus mux to choose r2, and set Add and LG=1 Clock cycle 3: FSM sets select lines on Bus mux to choose G, and set Lr0=1	

Nios II processor 32-bit registers	
r0	zero register
r1	assembler use
r2~r23	general purpose
r24~r3	special purpose
pc	program counter
r27/sp	stack pointer 0x20000
r28/fp	frame pointer
r29/ea	exception return address
r31/ra	return address

Machine Code (instruction bit pattern)

I-Type Instruction Word Format: Operation: arithmetic, logical, "branch", load and store, cache management (addi,andi)

31 — 5bits	27 — 26 — 5bits	22 — 21 — (16 bits)	6 — 5 — (6 bits)	0
Register field (A) source operand	**Register field (B)** destination	**immediate data field** source operand (signed,except for logic operations and unsigned comparison)		**Operation Code**

R-Type Instruction Word Format: Operation: arithmetic, logical,comparison, "custom" (add,nor,cmpeq,cmplt)

all register operands, all arguments& results specified as registers.

31 — 27	26 — 22	21 — 17	16 — 6	5 — 0
Register field (A) source operand	**Register field (B)** source operand	**Register field (C)** destination	**opcode-extension field (OPX)** sometimes a small IMM value in the 5 low-order OPX bits. OPX unused bits = 0.	**opcode field (OP)**

J-Type Instruction Word Format: Operation: "call", "jmpi" (transfer execution anywhere within a 256-MB range)

31 — 6	5 — 0
IMM26	**OP**

Instruction Set (operation):

ldw r2,4(r3)	# r2<-[memory at address r3+4]
(load word) read a word from an address in memory	
br LABLE	# branch to address, implemented as pc<-[pc]+offset (in byte)
offset: 2's complement value needed to reach the LABLE e.g END: br END # offset=-4	
srli r9,r9,1	# shift right
and r9,r9,r11	# bitwise AND
beq r9,r0,LABLE	# conditional branch
addi r9,r9,1	# bitwise AND
ret	# pc<-[r31] pc<-[ra]

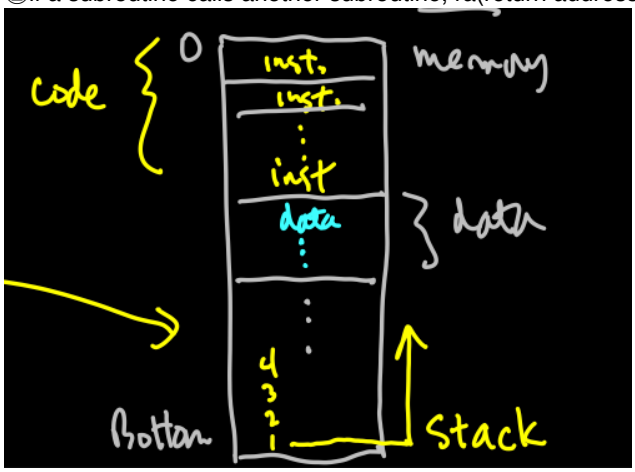
```
    .text
    .global
_start
_start:
      XXX
END: br END
    .data
NUM:
    .word  XXX
    .end
```

Subroutine and Stack:

stack: (First-In-Last-Out) (pushdown stack): a list of data elements(words) with accessing restriction that elements can be added or removed at one end of the list only.

①in subroutine, we should not change the contents of registers. (save regs in memory on the stack at the start of subroutine and restore them before return)

②if a subroutine calls another subroutine, ra(return address) will be wrong, dont ret to main (save ra, restore before ret).




```
ONES:  subi  sp,sp,12  //pre-decrement sp
       stw  r9,0(sp)
       stw  r11,4(sp)  //save
       stw  r31,8(sp)
//sp 始终指r9, sp的值(stack address)没有改变
//sp: pointer to an location in memory
LOOP:  XXXXXX
       br  LOOP
END_ONES:
       ldw r9,0(sp)
       ldw r11,4(sp) //restore
       ldw r31,8(sp)
       addi sp,sp,12
//sp 指向r31 下面的一个address
       ret
```

FINDSUM:

```
    .text
    .global _start
_start:
movia   sp, 0x20000
     add   r2, r0, r0        /* Clear the register= mov r2,r0 */
     ldw   r4, N(r0)   /* data is passed to subroutine in r4 */
     call   FINDSUM        /* result will be in r2 */
END: br   END          /* wait here */
FINDSUM:
     bne   r4, r0, RECURSE
     add   r2, r0, r0
     ret
RECURSE:
     subi   sp, sp, 8
     stw    r4, 0(sp)
     stw    ra, 4(sp)
     subi   r4, r4, 1
     call   FINDSUM   #recursion point  #break point is N==0 ret
     ldw    r4, 0(sp)
     add    r2, r4, r2
     ldw    ra, 4(sp)
     addi   sp, sp, 8
     ret
   .data
N: .word   2
   .end
```

```
/* bad code */
FINDSUM:
  subi  sp, sp, 8
  stw  r4, 0(sp)
  stw  ra, 4(sp)
  beq   r4, r0, END_LOOP #exit when N==0
  add   r2, r2, r4
  subi  r4, r4, 1
  br    LOOP
END_LOOP:
  ldw   r4, 0(sp)
  ldw   ra, 4(sp)
  addi  sp, sp, 8
  ret
```

I/O Device: memory mapped I/O is used to read/write from I/O device.

```
    .text
    .global _start
_start:
    movia r16,0x10000000  #r16->LEDR
    movia r20,0x10000020  #HEX3~0
    movia r15,0x10000040  #SW
    movia r17,0x10000050  #KEYs
    ldw   r6,HEX_BITS(r0) #initial pattern
DO_DISPLAY:
    ldwio r4,(r15)   # r4<-[SW]
    stwio r4,(r16)   # display on LEDR
 //r16(LEDR)<-[r4](SW)
    ldwio r5,(r17)   # r5<-KEYs
    beq r5,r0,NO_KEY
KEY: mov r6,r4  # use SW as pattern  //r6<-[r4](SW)
WAIT: # wait for key release
    ldwio r5,(r17)
    bne r5,r0,WAIT
NO_KEY:
    stwio r6,(r20) # write to HEX0~3
    roli r6,r6,1  # rotate
    ldw  r7,DELAY_VALUE(r0)
DELAY:
    subi r7,r7,1
    bne  r7,r0,DELAY
    br  DO_DISPLAY
   .data
HEX_BITS:   .word    0xf
DELAY_VALUE:    .word       100000
   .end
```

Timer for Exact Delay

```
_start:
    movia r16,0x10000000  #r16->LEDR
    movia r20,0x10000020  #HEX3~0
    movia r15,0x10000040  #SW
    movia r17,0x10000050  #KEYs
    ldw   r6,HEX_BITS(r0) #initial pattern
//load time into timer (r18 points to)
    movia r18,0x10002000 # r18->timer
    movia r12,2500000  # 50MHz×2.5e6=0.05s
    sthio r12,8(r18)  # load counter low half
    srli  r12,r12,16
    sthio r12,0(r18)  # load counter high half
    movi  r19,0b0110  # start=1,CONT=1    ??????????
    sthio r19,4(r18)  # counter is now running


                      Same AS LEFT


DELAY:
    ldhio r7,(r18) #read status register
    andi r7,r7,1
    beq  r7,r0,DELAY
    sthio  r7,(r18) #clean TO bit
    br  DO_DISPLAY
```

ldwio/stwio same as ldw/stw,except if there is a data cache on processor.
(load data from cache memory/IO device)
Interval timer: I/O device with a counter counting down to 0 at a known Clock rate.

Bubble sort

```
/* This program implements a bubble sort algorithm */
r20: How many Number to be sorted
r9: The list of Number to be sorted
r18: flag register
r19: element counter (make sure all number checked in one loop)
     r4: point to the 1 number to be sorted
     r5: the former number       r6: the latter number

     .text
     .global _start
_start:
     movia          sp, 0x20000
     movia          r9, LIST            /* Point to the start of the list */

BEGIN_SORT:
```

```
     ldw            r20, 0(r9)          /* r20 = size of the list */

RESTART_SORT:
     add            r18, r0, r0       /* Clear the flag register */
     addi           r19, r0, 1    /* Reset the loop counter */
     addi           r4, r9, 4     /* r4 points to the first element to be sorted */

SORT_LOOP:
     call           SWAP          /* swap the list elements if not sorted */🎓🎄or
r18, r18, r2     /* set flag according to return value from SWAP */
     addi           r19, r19, 1                /* increment the element counter */
     addi           r4, r4, 4           /* move to the next element in the list */
     bne            r19, r20, SORT_LOOP              /* loop through all the elements */
     addi           r20, r20, -1              /* no need to resort the last element! */
     bne            r18, r0, RESTART_SORT            /* was at least one swap done? */

END:
     br             END                 /* Wait here once the program i👉s done    */

/*swap list elements;r4 points to the first element; return 1 in r2 if swap performed */
SWAP:
     addi           sp, sp, -12
     stw            r5, 0(sp)                                          /* save */
     stw            r6, 4(sp)                                          /* save */
     stw            ra, 8(sp)                                          /* save */

     add            r2, r0, r0                /* initialize return value to 0 */
     ldw            r5, 0(r4)          /* get the first list element from memory */
     ldw            r6, 4(r4)                  /* get the second list element */
     bgt            r5, r6, SKIP_SWAP   /* are the list elements already sorted? */

     stw            r6, 0(r4)           /* swap the list elements */
     stw            r5, 4(r4)
     addi           r2, r0, 1           /* set return value to 1 */

SKIP_SWAP:
     ldw            r5, 0(sp)                                          /* restore */
     ldw            r6, 4(sp)                                          /* restore */
     ldw            ra, 8(sp)                                          /* restore */
     addi           sp, sp, 12
     ret

     .data
LIST:
     .word          10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
     .end
```

Divide

```
 .text
   .global _start
_start:
     movia    sp, 0x20000

     ldw         r4, A(r0)   # load operand A
     ldw         r3, B(r0)   # load operand B
     call     DIVIDE    # compute C = A / B
END:
  br    END    # wait here

/*********************************************************************************
 * DIVIDE - Implements a soft divide "instruction"
 * produces C = A / B,
 *    the most-sig halfword of C is remainder and the least-sig halfword is quotient
 * inputs: r4 = A, r3 = B                  * outputs: r2 = C
*********************************************************************************/
     .global  DIVIDE
DIVIDE:
     subi           sp, sp, 4                /* reserve space on the stack */
     stw            r4, 0(sp)
     add            r2, r0, r0             /* Clear the register */
DIV_LOOP:
```

```
      blt             r4, r3, FINISH
      addi            r2, r2, 1                   # increment quotient
      sub             r4, r4, r3
      br              DIV_LOOP
FINISH:
      # r2 has quotient, r4 has remainder
      slli            r4, r4, 16
      or              r2, r2, r4

      ldw             r4, 0(sp)
      addi            sp, sp, 4

      ret

      .data
A:
      .word           10
B:
      .word           3
      .end
```

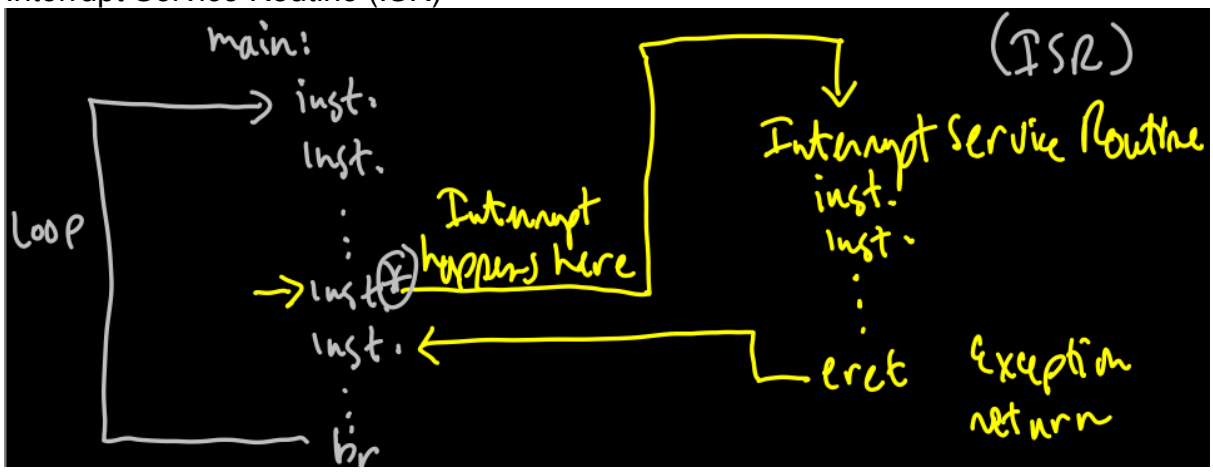
Interrupt

NiosII interrupt step:

IRQ occurs --> Nios II aborts the current instruction, and sets **ea** to the address of the next instruction, then Nios II copies **status** register into **estatus**. Then Nios II sets PIE to 0 in **status**. Nios II branches to 0x20 (exception address)

Interrupt Request: irq0~31 (from I/O device,like DataIn)

Interrupt Service Routine (ISR)



①Processor executed normally. During execution of Instruction*, interrupt signal received on irq(0~31). ② Instruction* is aborted. Processor auto branches to a specific address (0x20) ③code at 0x20 checks which interrupt occurred and then call appropriate ISR.

Note: we need to save all regs at the start of ISR using Stack and restore before eret(exception return).

6 control register: ctl0~5

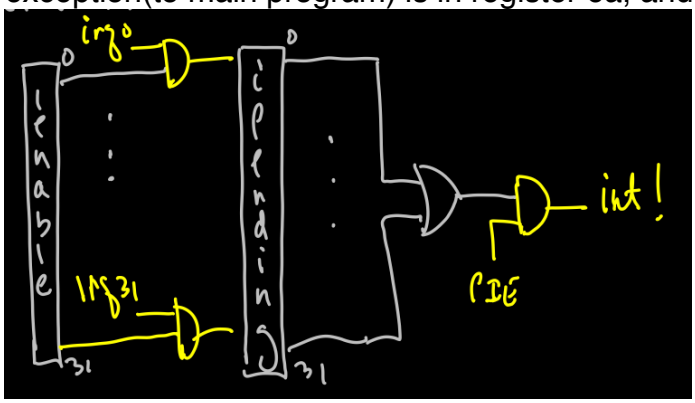
Processor does not respond to interrupts unless told to do so using control regs.

ctl0	status	PIE=1 allow interrupts	
31	2	1	0
not use			Processor Interrupt Enable(PIE) ✳

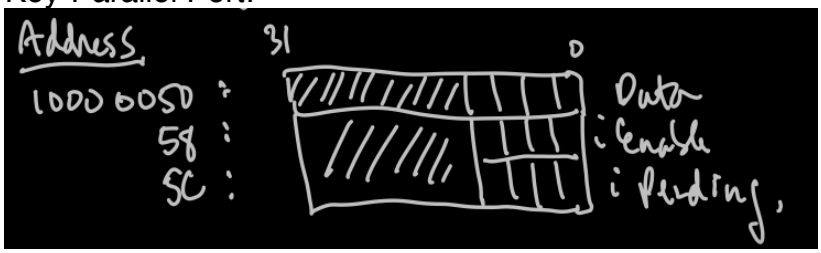
ctl1	estatus	when interrupt happens, ctl1<-[ctl0]
ctl2	bstatus	used for debugger
ctl3	ienable	bit 0 enables irq0...bit 31 enables irq31 { =1enable irqs}
ctl4	ipending	bit 0 =1 if irq0 interrupts is pending
ctl5		ignore

Exceptions:

Interrupt are special type of exception.
Other exceptions correspond to internal conditions in processor: divide by 0, illegal instruction.
Exception Processing: setting PIE bit in ctl0 to enable all irqs, and also setting individual irq enable in ctl3. It also use irq4 to see which irq happened. The return address for exception(to main program) is in register ea, and accessed by eret instruction.



Enable Interrupt Generation in I/O device:
Key-Parallel Port:



Exception Handler Code (address 0x20)

```
    .section  .exception,"ax"
    .global   EXCEPTION_HANDLER
EXCEPTION_HANDLER:
       XXXXX
```

memory layout with exceptions:
0: br _start (code executed when Reset of NiosII pressed)
0x20: code for figuring out which irq or other exception occurred, and calling the appropriate ISR or exception subroutine
_start 0x400 main program starts at an address high enough to leave sufficient space for the exception handler code.

interrupt: ① ea = pc +4 ②pc <-0x20 (exception handler address) ③ ctl1 = ctl0

Basic Functional Units of a Computer

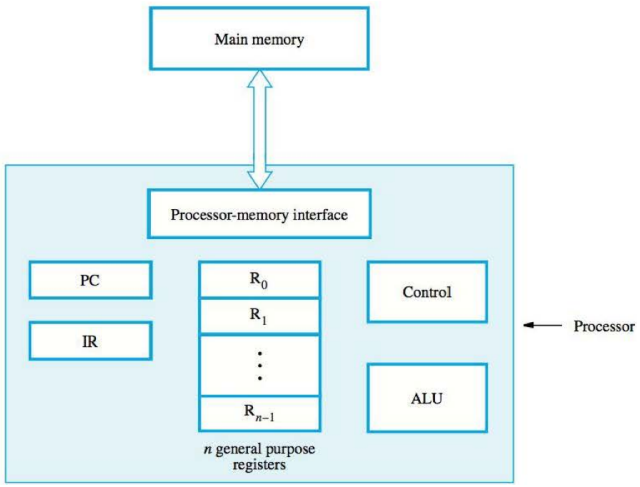
	memory	
Input →	(store programs and data)	←arithmetic and logic(ALU)
(accept coded information)	interconnection network	(execute operations)
output →		←control
(send processed results to outside world)		(sends control signals to other units and senses their states)
I/O		processor

information= instruction+data
program: a list of instructions which performs a task. (stored in the memory)
Data: numbers and characters used as operands by the instructions (stored in the memory)
instruction & data(number+character) is encoded as bits (a string of binary digits).
Bit have one of two possible values, 0 or 1(two stable states).
Memory (storage)
①Primary/Main Memory:(expensive, does not retain information when power turned off)
words: groups of semiconductor storage cells in fixed size (word length(number of bits in each word): 16, 32, 64 bits)
address: word location (consecutive numbers start from 0 & identify successive locations)
random-access memory (RAM): A memory in which any location can be accessed in a short and fixed amount of time after specifying its address.
memory access time: The time required to access one word. [ns] (indep of address)
cache Memory: hold sections of a program being executed currently, along with any associated data.
At the start of program execution, the cache is empty. All program instructions and any required data are stored in the main memory. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

②Secondary Memory: (inexpensive, permanent *storage*)
(access time longer) e.g *magnetic disks, optical disks* (DVD/CD), *flash memory devices*
ALU:
operands are brought into the processor, and stored in *registers(*high-speed storage elements*).*
Each register can store one word of data.
Access times to registers are even shorter than to the cache unit on the processor chip.
Control: generate the *timing signals* that govern the transfers(e.g data transfers between the processor and the memory) and determine when a given action is to take place.
I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred.

wire: (control lines) carry the signals used for timing and synchronization of events in all units.

Basic Operational Concepts



Instruction Register (IR): hold instruction that is currently being executed.
Program Counter (PC): contain/point to memory address of the next instruction to be fetched from the memory and executed
the contents of the PC are incremented so that the PC points to the next instruction to be executed
Processor/*general-purpose Registers* R_0 through R_{n-1} (PR)

Processor-Memory Interface: manage the transfer of data between the main memory and the processor.
Read/Write
handling I/O transfers:
interrupt: interrupt-service routine
execution of the current program must be suspended, its state must be saved in the memory before servicing the interrupt request(information saved includes the contents of the PC&PR, and some control information)
interrupt-service routine is completed, then the state of the processor is restored from the memory so that the interrupted program may continue.

Machine Instruction:

Load R2, LOC	Add R4, R2, R3	Store R4, LOC
This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2. The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps. First, the instruction is fetched from the memory into the processor. Next, the operation to be performed is determined by the control unit. The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2.	adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.	This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.

Parallelism:
①instruction-level parallelism(*pipelining*)

②multicore processor

③multiprocessor: shared memory multiprocessors

④Multicomputer: message- passing multicomputers

interconnected group of complete computers: access only to their own memory units and exchange *messages* over a communication network.

Instruction Set Architecture (ISA) of a processor

To execute a high-level language(C,C++,java) program on a processor, the program must be translated by a compiler program into the machine language for that processor .
Assembly language is a readable symbolic representation of machine language.

1. Memory Location and Addresses:

storage cell: store a bit of information having the value 0 or 1.

a word of information: group of n bits (n: word length, modern computer 16,32,64 bits)

signed number 32 bits=4 bytes character 8 bits=1 byte

Address: $0 \sim 2^k-1$ (2^k addresses constitute the address space of the computer)

1 mega=2^{20}=1048576 1 giga=2^{30}= 1 kilo=2^{10} 1 tera=2^{40}

<1.1> byte-addressable memory (byte address, word address)

<1.2> big-endian assignment: lower byte addresses are used for the more significant byte (leftmost byte) of the word

little-endian assignment: lower byte addresses are used for the less significant byte (rightmost byte) of the word

<1.3> word alignment: word locations have aligned addresses if they begin at a byte address that is a multiple of the number of bytes in a word.

2. memory operations:

① Read operation: transfers a copy of the contents of a specific memory location to the processor.

The memory contents remain unchanged.

To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

② Write operation: transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location.

To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

3. Instruction & Instruction Sequencing:

An instruction specifies an operation to be performed and the operands involved.

mnemonic: abbreviations of the words defining the operations.

instructions operational capablity:

①Data transfer between memory and processor register (PR) ②Arithmetic and Logic Operations on data ③ Program sequencing and Control ④ I/O transfers

<3.1> Register Transfer Notation:

R2←[LOC] (contents of memory location LOC are transferred into processor register R2)

R2←[R3]+[R4] (adds the contents of registers R2 and R3, and places their sum into register R4)

RH:value LH: name of a location where the value is to be placed, overwriting the old contents of that location.

R2:processor register LOC:address of memory location DATAIN: Register in the I/O subsystem

<3.2> Assembly-Language:

①the contents read from a memory location are loaded into a processor register

Load R2, LOC (contents of LOC unchanged, contents of register R2 overwritten)

②Add R4, R2, R3

<3.3> RISC and CISC instruction set: Reduced/Complex Instruction Set Computers ??

<3.4> Reduced Instruction Set Computers:

Characteristic:

Each instruction fits in a single word.

A load/store architecture is used, in which

- Memory operands are accessed only using Load and Store instructions.
- All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly with in the instruction word.

<3.5> instruction execution & straight line sequencing:

instruction register (IR): determine which operation is to be performed.

Program Counter (PC): hold the address of next instruction to be executed. [address]

straight-line sequencing: increasing addresses (During the execution of each instruction, PC is incremented by 4 to point to the next instruction)

instruction fetch/execute

<3.6> Branching

4. Addressing Mode: (data structure)

addressing modes: the way for specifying the locations of instruction operands

①Register Mode: The operand is the contents of a PR; name of register is given in the instruction.

effective address of the operand = R_i

②Absolute Mode: The operand is in a memory location; address of the location is given explicitly in the instruction.

effective address(EA) = LOC

③Immediate mode: The operand is given explicitly in the instruction.

Operand = a signed number

④Indirect mode: The effective address of the operand is the contents of a register that is specified in the instruction. (Ri)

EA=[Ri]

⑤Index mode: The effective address of the operand is generated by adding a constant value to the contents of a register. index X(Ri)

X:offset (displacement) from this address to the location where the operand is found.

EA=[Ri]+X

⑥ base register Rj contain the offset X： base with index (Ri,Rj)

EA=[Ri]+[Rj]

X(Ri,Rj) EA= [Ri]+[Rj]+X

X(R0) = absolute mode EA=X

5.Assembly Language：

mnemonic: instruction operations to represent the corresponding binary code patterns

source program: the user program in its original alphanumeric text format

object program: the assembled machine-language program

binary pattern, or operation (OP) code, for the operation performed by the instruction.
<5.1> assembler directives/commands:
Load R2,N   # load a number
Move R4, #NUM   # R4 stores the address of NUM1
Load R2,(R4)   # load the value R4 pointing to.

6. Stack:(data structure)
Stack (pushdown stack): a list of data elements(words) with accessing restriction that elements can be added or removed at one end of the list only. (Top: stack end & Bottom:)
storage mechanism: Last-In-First-Out(LIFO)
> push: place a new item on stack Subtract SP, SP, #4 Store Rj, (SP)

> pop: remove the top item from stack Load Rj, (SP) Add SP, SP, #4

Stack Pointer(SP): a PR points to processor stack.
stack grows in the direction of decreasing memory addresses
7. Subroutine:

Basic I/O